UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

APLICACIÓN MÓVIL PARA DISPOSITIVOS ANDROID PARA LA TRAZABILIDAD DE ACTIVOS

Alejandro Sastre Izquierdo
Tutor: Santiago Panizo Jiménez
Ponente: Gonzalo Martínez
MAYO 2016

APLICACIÓN MÓVIL PARA DISPOSITIVOS ANDROID PARA LA TRAZABILIDAD DE ACTIVOS

AUTOR: Alejandro Sastre Izquierdo
TUTOR: Santiago Panizo Jiménez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2016

Resumen (castellano)

Este Trabajo Fin de Grado está desarrollado con la intención de dar a conocer los métodos de trabajo y desarrollo que se utilizan en las empresas para grandes proyectos. En concreto, se realizará a partir de la experiencia obtenida en la empresa de Athelia con el proyecto de Servitrax, uno de los principales proyectos dentro de la empresa.

El proyecto consistirá en el desarrollo de una aplicación Android utilizando la plataforma de Xamarin. Esta plataforma nos permitirá desarrollar aplicaciones Android utilizando C# en lugar del lenguaje nativo de Android que es Java. Esta aplicación destaca por ser una aplicación muy completa con bases de datos, acceso web y una lógica de negocio muy compleja. La aplicación estará orientada a la trazabilidad de activos y productos, es decir, servirá para mantener un control de los productos y activos de Air Liquide (cliente que contrata el proyecto). Esta aplicación surge con el objetivo de sustituir la aplicación actual que lleva más de 10 años en producción y que está desarrollada para la plataforma de Windows Mobile.

A lo largo de este documento se explicará la forma de trabajar, las metodologías usadas (Scrum) y los principios y técnicas del software que se han seguido (cómo por ejemplo TDD o SOLID). Además, se explicarán las herramientas que se han utilizado, cómo se ha organizado y gestionado el trabajo en equipo y cómo se han realizado las entregas y las pruebas.

También veremos ejemplos del código realizado, centrándonos sobre todo en las técnicas usadas y en los componentes que se han desarrollado. Estos componentes nos permitirán agrupar funcionalidad usada en varias partes de nuestra aplicación en un solo componente que facilite su uso. Dentro de la sección de desarrollo, veremos también algunas pantallas explicadas más en profundidad y el procedimiento usado para realizar los test que será fundamental para la lógica de negocio.

Al final del documento, encontraremos dos anexos: El primero explicará las principales diferencias entre el desarrollo Android con Java y el desarrollo con Xamarin. El segundo explica qué información y manuales necesita un programador para poder empezar a desarrollar en el proyecto.

Abstract (English)

This Bachelor Thesis has been developed with the idea of sharing the work and development methods that the companies use for big projects. Specifically, this document will be developed from the experience obtained working for Athelia in the Servitrax project, one of the biggest projects of the company.

This project will consist in the development of an Android application using the Xamarin platform. This platform will allow us to develop Android applications using C# as native code instead of Java. This application is remarkable for being a complete Android application, using databases, web access and a very complex business logic. The application will be focused on the traceability of the assets and products, in other words, it will be used to keep track of the products and assets of Air Liquide (client's project). This application arises from the necessity of replacing the former application which has been running the last 10 years and that has been developed for the Windows Mobile OS.

Along this document, it will be explained the work methods, the methodologies used (Scrum) and the principles and software techniques that has been applied (like TDD and

SOLID). In addition, it will be explained the tools used, and how the schedule, the deliveries and the test have been done.

This project will also explain project examples, focusing in the techniques used and the components that have been developed. These components will allow us to group functionality used in multiple places, in generic components easier to use. Inside the development section, we will see details of some of the screens, and the test development which will be a crucial part in the business logic development.

At the end of the document, there will be two appends: The first one will explain the main differences between Android Java development and Android Xamarin, the other one will explain the guides and information that any new programmer that starts with the project is going to need to start developing.

Palabras clave (castellano)

Xamarin, Android, C#, Trazabilidad, Trabajo en equipo, Mantenimiento software, SCRUM, TDD

Keywords (inglés)

Xamarin, Android, C#, Traceability, Teamwork, Software maintenance, SCRUM, TDD

Agradecimientos

A mis compañeros de trabajo que tanto me han enseñado, a mis amigos que me han acompañado a lo largo de toda mi carrera y a mi familia que me ha hecho ser quien soy.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

En el presente documento, procedo a detallar el trabajo que he estado realizando durante mi tiempo en Athelia Solutions como miembro del proyecto Servitrax Android (SVX Android). Este proyecto consiste en la renovación completa de la aplicación actual existente en Windows Mobile. Athelia ha formado parte del proyecto casi desde los comienzos y podemos afirmar que es uno de los proyectos más importantes de la empresa. Tras estar 4 meses trabajando con ellos, me introdujeron en este proyecto que justo en ese momento comenzaba a desarrollarse. Esta memoria contiene la explicación del software desarrollado, las metodologías usadas y los componentes desarrollados durante este tiempo. Estos han sido elaborados con el objetivo de conseguir una aplicación Android, fácil de mantener, y de larga duración. Como base para el desarrollo hemos usado Xamarin que nos da la capacidad de desarrollar Android utilizando el lenguaje y las tecnologías de C#.

Athelia AIR LIQUIDE Creative Oxygen Servitrax™

Esta aplicación estará desarrollada para cuatro entornos diferentes (Canadá, MyGasPartner, Sudáfrica y Europa) y estará presente en más de 10 países con más de 3500 terminales y más de 10000 usuarios activos.

1.1 Motivación

1.1.1 Motivación de la empresa

La aplicación a desarrollar surge con el objetivo de renovar todo el software que forma parte del proyecto de Servitrax. Parte de este proyecto, es la aplicación desarrollada en Windows Mobile y que está en la actualidad en producción, esta aplicación se ha evolucionado, ampliado y mantenido durante unos 10 años con lo que se ha convertido en una aplicación difícil de mantener al tener una gran complejidad y tamaño. Además, está desarrollada para unos dispositivos que se están dejando de fabricar (terminales Windows Mobile). Por tanto, este desarrollo tiene el objetivo de mejorar la aplicación actual utilizando un SO moderno, en este caso Android. Dentro de Android, se decidió desarrollar sobre C# utilizando Xamarin, en parte para poder reutilizar parte de la lógica presente en la aplicación actual y, en parte, debido a la amplia experiencia ya existente en la empresa con C#. Además, aparte de la renovación de la aplicación en los terminales móviles, otro equipo de trabajo, procederá a realizar a lo largo de este año y del siguiente, la actualización del servidor, de otro cliente pesado basado en Windows y del subsistema de integración con el ERP de la compañía, renovando todo el conjunto que forma el proyecto de Servitrax. Debido a que la aplicación Android acabará de desarrollarse antes que el nuevo servidor, nuestra aplicación deberá ser compatible con el servidor antiguo y con el nuevo una vez desarrollado. Con este objetivo, nuestro equipo tendrá la tarea de desarrollar la aplicación Android y un intermediario (llamado *wrapper*) que usará la interfaz del nuevo servidor utilizando la nueva Api, pero redirigiendo las conexiones al servidor antiguo realizando las transformaciones necesarias.

1.1.2 Motivación personal

En el ámbito personal, yo decidí participar en este proyecto porque me permitía seguir aprendiendo y mejorando mi desarrollo Android, además, me permitía aprender a diseñar proyectos de grandes dimensiones con un plan de mantenimiento a largo plazo (se espera

que esta aplicación se mantenga en uso los próximos 10/15 años) y, me permitía profundizar en el desarrollo de aplicaciones con Xamarin. Considero que este trabajo pone en práctica los conocimientos aprendidos a lo largo de mi carrera y que, es interesante, la forma de trabajar en una empresa de software, su forma de probar correctamente un software dirigido a un cliente y conocer las decisiones tomadas en un proyecto de estas dimensiones. Desarrollando esta aplicación he aprendido a aplicar correctamente patrones de diseño, a aplicar principios de diseño orientado a objetos (SOLID) y a gestionar proyectos de gran envergadura utilizando herramientas de control de versiones, de planificación y supervisión (Jira), de colaboración de equipos (Confluence), de integración continua (TeamCity) y metodologías de desarrollo como Scrum, TDD o Pair Programming.

1.2 Organización de la memoria

El presente documento, muestra los pasos realizados hasta ahora para la elaboración de este proyecto, centrándose más en las partes realizadas por mí. Este documento, además de la explicación de cómo se ha realizado la aplicación, incluye las decisiones de diseño que hemos tomado a lo largo del proyecto, las aplicaciones que se han usado para la organización, y las herramientas que se han usado en la elaboración del mismo. La estructura del documento es la siguiente:

- Estado del arte:
    - Descripción de la empresa.
    - Proyecto de Servitrax, en qué consiste, qué partes tiene, qué usuarios la van a utilizar y en qué situaciones, ...
    - Herramientas de desarrollo utilizadas en el proyecto, tanto para desarrollo como para la organización del equipo.
- Diseño:
    - Cómo se ha planificado el proyecto y cómo son las etapas en las que se divide.
    - Aspectos que se han tenido en cuenta para definir la interfaz.
    - Qué metodologías y técnicas se han aplicado.
    - Cómo se ha estructurado el proyecto, para mantener separadas la capa de negocio y aplicación y por qué esto es muy importante para un correcto mantenimiento de la misma.
    - Pruebas de concepto que se han realizado para tener claro antes de empezar qué técnicas y elementos usar.
- Desarrollo:
    - Proceso seguido en la preparación del proyecto y la preparación del entorno de trabajo.
    - Cómo es la metodología de desarrollo para desarrollar cada tarea de la aplicación.
    - Principales componentes que se usan en varios puntos de la aplicación.
    - Algunas actividades y servicios de la aplicación que destacan por su relevancia o complejidad.
- Integración, pruebas y resultados:
    - Cómo se realizan las pruebas, los documentos a realizar para el cliente y cómo se prepara la entrega del producto realizado.
- Conclusiones y trabajo futuro:
    - Analizaremos y explicaremos todo el trabajo que queda por hacer, la planificación futura, y conclusiones sobre lo que ha supuesto el proyecto.

- Finalmente, se podrán encontrar referencias a los documentos en los que nos hemos basado a lo largo de todo el proyecto, un glosario y anexos con información adicional entre los que destaca, una explicación más detallada de las diferencias entre Xamarin y el desarrollo con Java.

2 Estado del arte

Para entender el proyecto, hay que entender a qué se dedica Athelia y obtener una visión más global del proyecto de Servitrax.

2.1 Athelia

Athelia Solutions Iberica es una empresa que, aplicando soluciones digitales, ofrece a otras empresas mejorar el rendimiento de su negocio, ayudando a las empresas a conocer con detalle sus productos y activos, y llevar actualizado un inventario de todos los elementos producidos por la empresa. Todo ello, utilizando herramientas como Big data, NFC, Internet de las cosas, Android u Oracle. En otras palabras, ayuda a las empresas a aplicar las nuevas tecnologías a sus industrias para controlar el flujo, la localización y estado de los activos que tienen en producción y, ayuda a los operarios a realizar sus tareas de una forma más efectiva.

2.2 Servitrax

El proyecto de Servitrax, es un proyecto de *Air Liquide* que surge con la idea de usar un sistema único de identificación para todas sus botellas e instalaciones. Este sistema dota de trazabilidad a los contenedores retornables de la compañía y optimiza los procesos de llenado, distribución y recuperación de sus productos.

Para una compañía como el grupo Air Liquide, el control de sus activos supone una importante mejora respecto a sus competidores, en concreto, la trazabilidad de sus contenedores retornables (en su mayor parte botellas de gas) supone anualmente un ahorro de millones de Euros en su cuenta de resultados. Dicho control, permite la optimización del número de contenedores sin tener que hacer constantes inversiones recurrentes.

Este proyecto, está formado por un conjunto de proyectos dedicados a la obtención y manipulación de los datos representados en el siguiente diagrama:

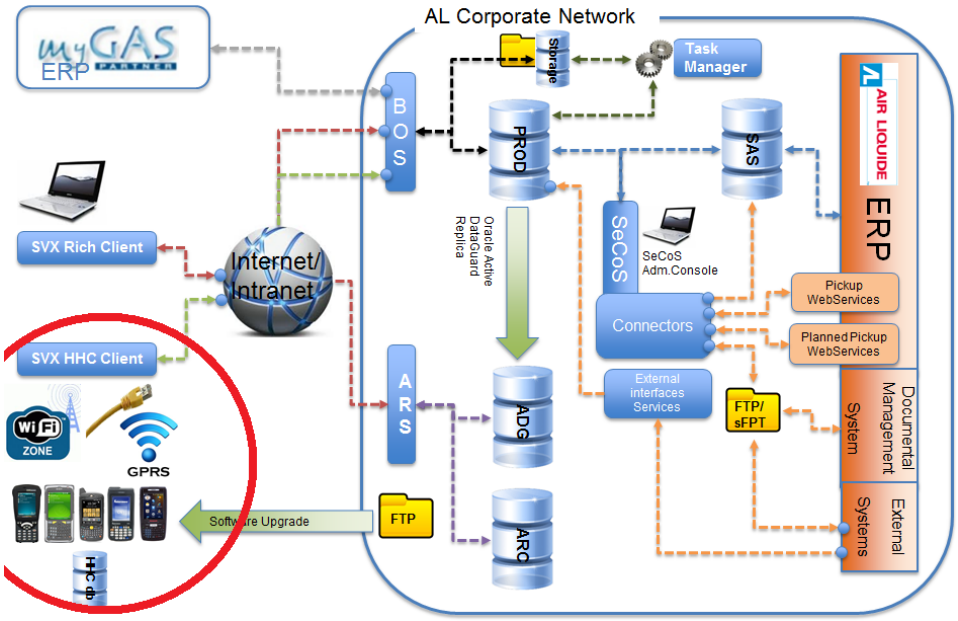


Figura 2-1 : Arquitectura Servitrax

Cómo vemos, aparte de nuestra aplicación en el dispositivo del usuario, existen también otros componentes como un cliente pesado para Windows (*Rich Client).*
Este cliente, está destinado a realizar tareas como controlar, revisar y editar pedidos y llenados de botellas, asignar envíos a usuarios, o revisar el proceso realizado por un distribuidor, pudiendo sacar informes de todas estas tareas. Es una herramienta de gestión de los activos de Servitrax, utilizada tanto en plantas de llenado como desde los centros administrativos de la compañía.
Aparte de este *Rich Client*, dentro de la red de AirLiquide (AL), encontramos el servidor central (Legacy Bos), y que provee al sistema tanto de las reglas de negocio aplicadas por la compañía, como la persistencia de la información.
Otro módulo es el llamado SeCos, encargado de importar y exportar a los ERPs información de negocio como pedidos, albaranes, rutas realizadas o la localización de activos.
Todo esto se complementa con un módulo de gestión de informes(ARS) y un módulo de archivado para almacenar los antiguos ciclos de vida de los activos.
Dentro de este diagrama, nuestro proyecto, Servitrax Android, corresponde con el Cliente de movilidad, y se centra en la obtención de datos por parte de los operarios, y la capacidad de identificar los contenedores y productos según van circulando por su ciclo de vida, notificando al servidor su estado y aplicando las reglas de negocio apropiadas. El ciclo de estos contenedores o botellas quedaría así:

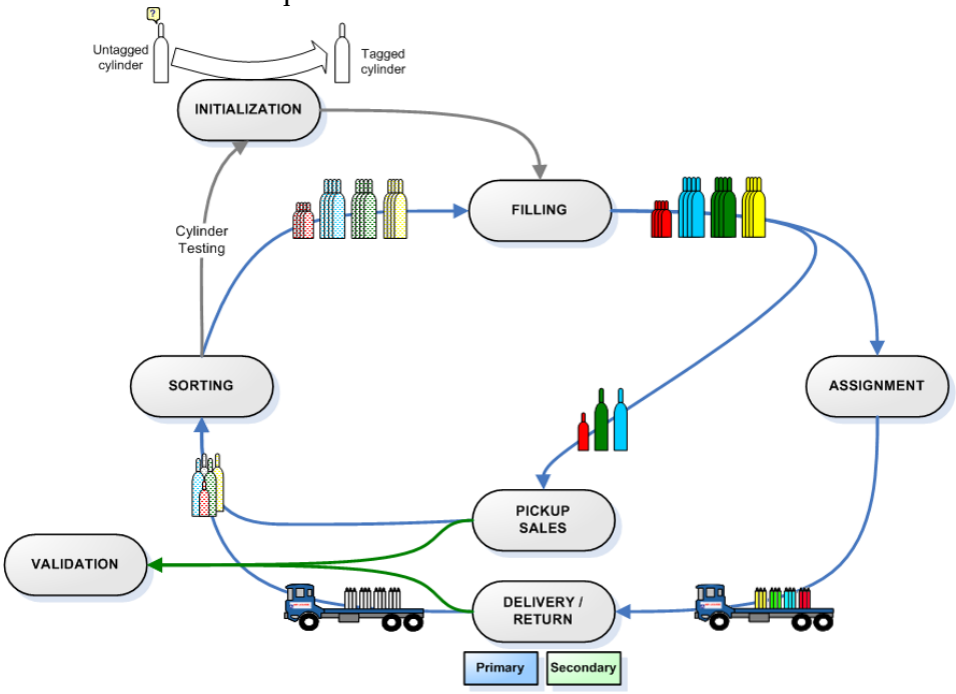


Figura 2-2 : Ciclo de vida de las botellas

Podemos identificar las siguientes fases:

- **Inicialización:** En esta fase se registra una botella para introducirla en el sistema. Para ello, se le asigna un identificador o código de barras y se añaden las características de la botella.
- **Llenado:** En esta fase se llena la botella con sus características correspondientes, además, se guarda información sobre cuándo se ha llenado, dónde y con qué producto.
- **Asignación:** También conocido como reparto, corresponde con la fase en la que se determinan que productos van a ir destinados a qué envíos. Asegurando que los

envíos van con las cantidades correctas de los productos y disminuyendo así, costes y problemas.

- **Entrega:** Es la parte en la que nos centramos en esta primera fase de la aplicación, está formada por dos tipos de entregas, los envíos a otras fábricas (distribución primaria) y los envíos a clientes y puntos de venta (distribución secundaria). Cada viaje (shipment) estará formado por un conjunto de notas que representan a los diferentes pedidos y clientes que se atienden durante este viaje. Además de entregar botellas, también se recogerán aquellas botellas que el cliente haya vaciado o dejado de usar con el objetivo de que vuelvan al ciclo de vida.
- **Recogida:** Se produce cuando un cliente va directamente a la fábrica a recoger un producto, no son planeadas por lo que no se pueden asignar productos a estos pedidos.
- **Validación:** No corresponde con el ciclo de las botellas, corresponde con la validación por parte del supervisor de las tareas realizadas por los usuarios, esta tarea será la única que no se realice en la aplicación Android, sino que será realizada desde el *Rich Client*.
- **Ordenamiento:** El objetivo de esta tarea, es agrupar en bloques de productos iguales aquellos productos que se han recogido del cliente. Además, durante esta fase, se podrá eliminar productos del ciclo de vida para que sean probados y se decida si pueden continuar en uso.

Nuestra aplicación está diseñada para que acabe valiendo para todas y cada una de estas fases excepto para validación, será utilizada por los operarios en las fábricas que llenan las botellas , los que asignan dichas botellas a los distribuidores tanto en distribución primaria como secundaria, los camioneros que las distribuyen, los operarios que las venden directamente, y los que las recogen y clasifican para decidir si dichas botellas pueden seguir usándose o deben ser revisadas y re-fabricadas en caso de defectos.

Por tanto, esta aplicación será usada en su mayoría, por operarios y camioneros de *Air Liquide*, en interiores como fábricas y almacenes, y en exteriores como es el caso del camionero que entrega el pedido al cliente. Habrá usuarios trabajando con equipos de producción industrial en ambientes ruidosos y con fuertes restricciones de seguridad como zonas con restricciones especiales. Además, al permitir el escaneo de código de barras para identificar los productos, se necesitará de terminales que cuenten con la capacidad de escanearlos.

En definitiva, al ser un conjunto de usuarios concreto y con terminales dedicados, esta aplicación estará específicamente diseñada pensando en sus necesidades.

El terminal que será usado por la mayoría de los usuarios y, por tanto, el que se ha usado para realizar las pruebas es el **Motorola TC55** [3]. Es un terminal orientado al entorno profesional y que cuenta con las siguientes características:

- Android 4.3 (API 18)
- Pantalla WGA
- Pantalla capacitiva y resistiva para soportar la manipulación de la aplicación con guantes.
- Garantía de protección IP67 y Gorilla glass 2.0
- Chip NFC
- Lector e imager integrados (lectores código de barras y QR)
- Batería de 2940 mAh ampliable a 4410 mAh.

Figura 2-3 : Motorola TC55

Pese a ser este terminal el que, con casi total seguridad, va a ser utilizado por los usuarios, nuestra aplicación está diseñada para soportar terminales desde la API 15 (Android Ice Cream

Sandwich 4.0.3) en adelante, soportando así el 97,7% de los terminales Android que existen actualmente en el mercado. Es importante destacar que los terminales son un activo que se reaprovecha, Dentro de una planta, varios usuarios comparten un terminal para diferentes funcionalidades.

2.3 Herramientas

A lo largo del desarrollo del proyecto, haremos uso de las siguientes herramientas:

- **Subversion (SVN):** Sistema de control de versiones, permite mantener un registro de las diferentes versiones del código, permitiendo comparar las diferentes versiones que se han ido realizando del proyecto y, facilitando el desarrollo en equipo. Además, permite la creación de *branches* a partir del proyecto que está en producción, sin afectar a los desarrollos en curso. Esta herramienta la usaremos combinada con Jira para mantener la trazabilidad del código fuente.
- **Jira:** Herramienta online desarrollada por Atlassian que nos permite repartir y controlar el desarrollo de las tareas definidas en la fase de diseño, revisarlas, y llevar cuenta del tiempo invertido por cada integrante del equipo en cada tarea. Esta herramienta estará conectada con SVN para sólo permitir subir cambios al servidor cuando van asociados a una tarea de la cual el usuario está a cargo. Cómo veremos en la fase de desarrollo y pruebas, esto nos será muy útil, permitiéndonos saber en todo momento qué partes del código han sido modificadas en cada tarea y quien ha sido el responsable. Además, proporcionará ayudas para el desarrollo de metodologías ágiles, permitiendo planificar iteraciones, elaborar informes y permitiendo a la dirección del proyecto supervisar el desarrollo de los integrantes de su equipo.
- **Confluence:** Herramienta para la colaboración de equipos desarrollada también por Atlassian, que nos permite compartir y editar la documentación de los proyectos, la usaremos también para mantener, tanto las especificaciones funcionales del proyecto, como documentos técnicos internos para el desarrollo del proyecto (notaciones, estructuras, técnicas, uso de componentes reutilizables…).
- **TeamCity:** Soporte para la integración continua, es un servicio que nos permite la generación automática de versiones, compila los diferentes proyectos, y comprueba los test existentes en los proyectos. Con la ayuda del plugin **Coverage**, nos permitirá también analizar la cobertura de los test unitarios y obtener algunas métricas de calidad de software.
- **Xamarin:** Es una plataforma de desarrollo que nos permite desarrollar código Android utilizando C# y las librerías .Net. Al final del documento, hay un documento donde se compara el desarrollo Android con Java respecto a C#.
- **Visual Studio:** Entorno de desarrollo creado por Microsoft para Windows. Será el IDE que usaremos para programar porque ofrece un gran soporte para C#. Se seleccionará este IDE sobre Xamarin Studio porque en el momento de analizar ambos IDE's, VisualStudio ofrecía un mejor soporte y gestión de los proyectos mientras que Xamarin Studio era todavía un software inmaduro. Además, tras la compra de Xamarin por parte de Microsoft, es altamente probable que Microsoft descontinúe el soporte de Xamarin Studio para centrarse en este IDE.

- **StyleCop:** Es un complemento de VisualStudio que analiza el código para controlar que se desarrolle acorde a unas reglas. Son reglas sobre la localización de los métodos dentro de una clase (públicos antes que privados, etc), reglas que obligan comentar todas las variables y métodos, reglas sobre la convención de nombres (Ej: métodos empiezan con mayúscula), y muchas más reglas que consiguen que el código quede comentado, limpio y estructurado de forma uniforme sin importar el desarrollador. Es una parte obligatoria dentro del desarrollo del proyecto y, como veremos en la parte de pruebas y revisión, es causa suficiente para tener que reabrir una tarea. Con el mismo objetivo, el de mantener una estructura común en todas las clases a lo largo de todo el proyecto, aparte de usar StyleCop, usaremos la plantilla de la Tabla 2-1 que organizará el código en regiones que luego en VisualStudio simplificarán la vista del código.

Tabla 2-1 : Plantilla para Clases

```
namespace $NAMESPACE$
{
    /// <summary>
    /// $CLASS_DESCRIPTION$
    /// </summary>
    public class $CLASS$
    {
        #region Constants
        #endregion

        #region Private Fields
        #endregion

        #region Lifetime
        #endregion

        #region Properties
        #endregion

        #region Public Methods
        #endregion

        #region Private Methods
        #endregion
    }
}
```

- **Oracle:** Es el sistema de bases de datos desarrollado por Oracle. Si bien no lo usaremos directamente, porque las bases de datos de Android están basadas en SQLite, si lo usaremos indirectamente porque el servidor utiliza este tipo de bases de datos y, por tanto, serán donde introduzcamos nuestros datos y realicemos consultas para realizar nuestras pruebas.
- **SQLite:** Es el sistema de bases de datos usado por Android para crear y gestionar bases de datos. Todas las bases de datos que creemos en el terminal usarán este sistema.
- **Entity Framework con SQLite:** En nuestro proyecto hemos usado Entity Framework para diseñar las bases de datos y el acceso a las mismas. En principio, Entity Framework no generaba bases de datos de SQLite por lo que hemos tenido que diseñar y modificar las plantillas de generación de dichas bases de datos.
- **NUnit:** Framework para test unitarios que originariamente estaba basado en cppUnit, posteriormente en JUnit y que ofrece la misma funcionalidad, pero centrada en C#. Será el sistema de pruebas que usaremos para probar toda la lógica de la aplicación presente en la capa de negocio.

3 Diseño

El diseño es una de las partes más importantes de un proyecto Software, en esta fase del proyecto se define la forma en que se va a desarrollar, las tareas en que se va a subdividir, las metodologías y técnicas que se van a usar, el estilo que va a tener la aplicación y, se hará un estudio de las posibles soluciones a los problemas más complejos.

3.1 Planning

El desarrollo del proyecto está divido en iteraciones, normalmente estas iteraciones se realizan de 2 a 3 semanas con una fase de desarrollo y una fase de pruebas entregando directamente al cliente final. En nuestro caso, tras cada iteración entregaremos a un equipo de control de calidad que realiza pruebas funcionales, comprueba que se cumplen los requisitos y entrega al cliente. Por tanto, para evitar que tanto nosotros como el equipo de pruebas estemos siempre probando, hemos espaciado más las iteraciones (hasta 5 semanas). En nuestro caso, al cabo de un conjunto de aproximadamente 4 iteraciones, generamos una Release Candidate que dicho equipo lleva al cliente, el cliente da sus opiniones y pone en preproducción dicha versión. De esta prueba, salen iteraciones para mejorar la aplicación y corregir errores.
Dentro de cada iteración, dividimos el proyecto en tareas para poder distribuir el trabajo entre los integrantes del equipo. Las tareas generalmente, tienen una duración entre 1 día y una semana porque en una tarea de menos de 1 día, la tarea tiene un gasto administrativo (informes, revisiones, ...) muy grande en comparación con la tarea en sí, y una tarea de más de una semana, consideramos que se puede repartir en tareas más pequeñas que faciliten el diseño y la planificación.
Para gestionar las tareas haremos uso de Jira, donde podremos ver y planificar las tareas que tenemos asignadas, indicar en qué hemos estado trabajando, revisar tareas realizadas por nuestros compañeros e incluir información adicional como capturas de pantalla y *release notes* para entregar al cliente. Permite, por tanto, una pequeña gestión del ciclo de vida del software para cada tarea.
Como podemos ver en la Figura 3-1, las tareas en Jira, se gestionan igual que en un Scrum panel, el desarrollador puede mover las tareas a los estados por lo que va pasando (Pendiente, En Proceso, To review, Terminado).

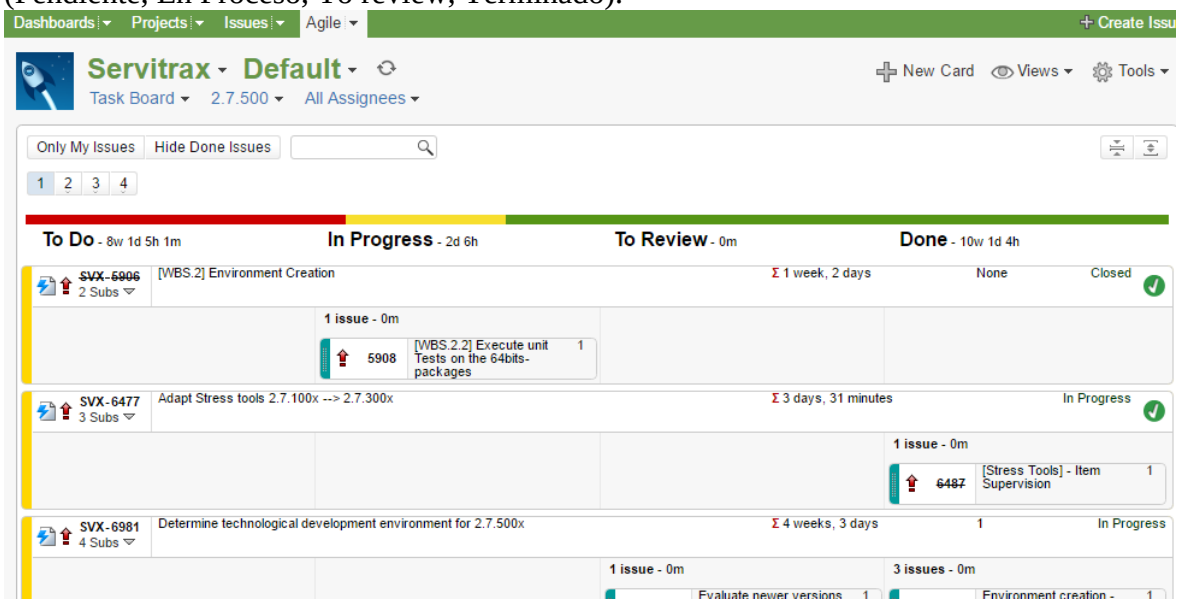


Figura 3-1 : Scrum Panel Jira

Será durante la fase de diseño, cuando se analicen los requisitos de la aplicación junto al cliente, y conforme a ellos, se irá dividiendo el proyecto en iteraciones y tareas. Las tareas estarán formadas por uno o más requisitos definidos en los documentos con los requisitos funcionales. El jefe del proyecto, junto con el arquitecto, serán los encargados de estimar la duración de cada tarea, y de ir asignando a cada miembro del equipo una tarea conforme se vaya avanzando en el desarrollo. Algunos conjuntos de tareas se juntarán en hitos para indicar que todas ellas pertenecen a una funcionalidad general.

3.2 Diseño de la interfaz

Para el diseño de la interfaz de usuario, el cliente contratará a un diseñador externo a nosotros, aun así, durante el desarrollo del proyecto mantendremos un contacto constante con el cliente para ir aclarando todos aquellos problemas que vayamos teniendo y para mostrarle las modificaciones en algunas pantallas, estas modificaciones, son debidas a que no son posibles de realizar (al menos en un tiempo razonable) en Android, a problemas de usabilidad o a cambios en los requisitos. Este contacto constante con el cliente, nos servirá también para ir aclarando aquellos requisitos que encontremos mal definidos o contradictorios.

A la hora de diseñar y realizar las pantallas de la aplicación (acciones, iconos, …) se tendrán en cuenta criterios de usabilidad como:

- **Sencillez:** La aplicación tiene que tener unos iconos sencillos, que indiquen claramente su utilidad, y hay que evitar procesos complejos a la hora de realizar tareas.
- **Diseño intuitivo:** Sobre todo en aquellas acciones que sean de efecto inmediato, se intentarán evitar acciones poco intuitivas, o acciones que conlleven resultados no esperados.
- **Diseño adaptado:** Entre los usuarios que van a usar la aplicación, nos encontramos con tres grupos que van a obligar a la aplicación a estar adaptada:
    - **Usuarios con guantes:** Además de utilizar terminales especiales, la interfaz deberá estar adaptada al grosor de los dedos con guantes.
    - **Usuarios con capacidad visual disminuida:** Algunos operarios sobre todo aquellos con edad avanzada presenta una capacidad visual disminuida.
    - **Usuarios que trabajan al aire libre:** La interfaz deberá estar adaptada para poder verse bien en condiciones de mucha iluminación (bajo el sol).
- D**iseño con "feedback":** Toda acción realizada por el usuario deberá tener una respuesta, y aquellos procesos que duren demasiado, deberán indicar que se están realizando.

Todos estos requisitos afectarán al diseño de forma que:

- Será necesario usar iconos con un diseño que indiquen lo que hacen de una forma clara.
- Será necesario usar colores contrastados con el fin de que se vea bien bajo condiciones de gran iluminación.
- Será necesario usar textos claros y grandes para facilitar la lectura.
- Se evitarán acciones complejas de lo normal, como deslizar, ya que son acciones que no se controlan bien con guantes y que en general no son intuitivas.

Además de estas decisiones, otra decisión influenciada directamente por la usabilidad es la limitación de la aplicación a sólo orientación vertical, esto facilitará mucho el diseño de la interfaz y el uso de la aplicación.

3.3 Metodologías

3.3.1 Scrum

Será la metodología utilizada para organizar el trabajo en equipo dentro del proyecto. Esta metodología pertenece al grupo de metodologías ágiles, en general, estas metodologías se caracterizan por un diseño incremental del proyecto, donde se van realizando entregas parciales al cliente [1]. Esta metodología nos permite mantener un control de los gastos del

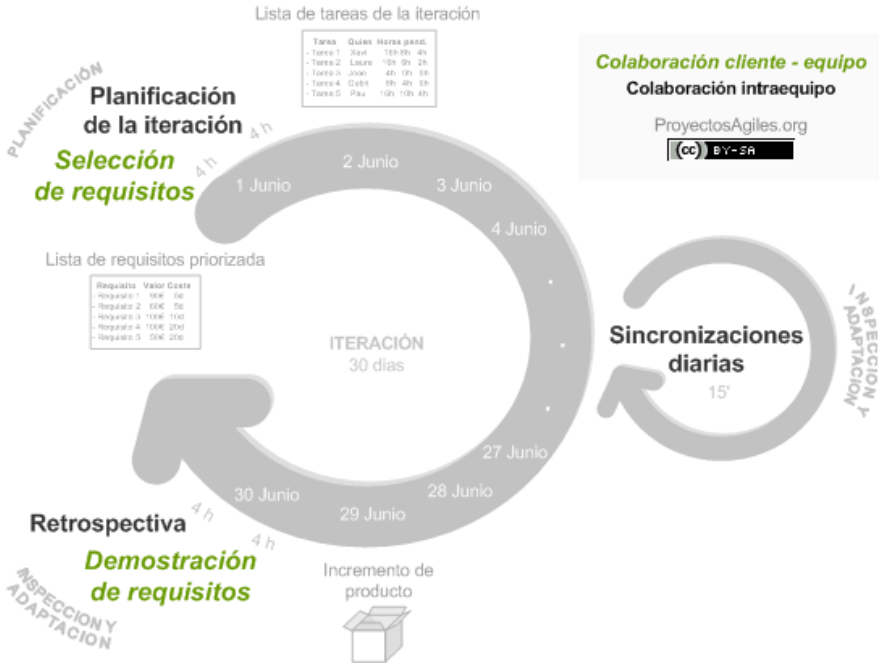


Figura 3-2 : Metodología Scrum

proyecto, un constante contacto con el cliente y la flexibilidad necesaria para poder reaccionar a cambios en los requisitos. Siguiendo esta metodología, cada día tenemos "reuniones" de los integrantes del equipo donde nos contamos lo que hemos estado haciendo el último día, qué vamos a hacer hoy y qué problemas hemos tenido o vamos a tener. Estas reuniones nos ayudan a conocer el proyecto en todo su conjunto y facilitan la comunicación con el equipo a la hora de solucionar un problema.
Aparte de estas reuniones, normalmente, una vez por iteración, tenemos reunión con el jefe de proyecto que hace las veces de Scrum Master. En ellas, nos presentan un resumen de cómo vamos con respecto a la planificación, nos solucionan problemas que tengamos que afecten al desarrollo del proyecto, y planificamos la próxima iteración definiendo nuevos objetivos.

3.3.2 TDD

Test Driven Development, es el desarrollo orientado a pruebas. Se caracteriza porque en lugar de empezar el desarrollo por las funciones que realizan la funcionalidad, se empieza por los test que prueban dicha funcionalidad. Al desarrollar primero los test, obliga al programador a centrarse en el diseño y en los casos de uso de dicha funcionalidad, cuándo se va a usar, qué salida se espera dada una entrada, y qué tipo de errores se esperan cuando falle. Este sistema consigue desarrollar código altamente reutilizable, con sólo lo necesario y con pocos fallos, además, facilita el trabajo en equipo, y convierte los test en el pilar de la aplicación. Esta metodología la usaremos sobretodo en la parte de negocio que será probada mediante NUnit.

Otra ventaja de TDD es el mantenimiento del proyecto porque permite:

A. Evaluar el impacto que posibles modificaciones tienen sobre el desarrollo, porque nos permite hacer pruebas de concepto de dichas modificaciones y medir el número de test que van a verse afectados al hacer un cambio en un método o clase y, calcular de este modo, el coste de realizar dicho cambio o mejora.

B. Evita efectos colaterales indeseados al realizar modificaciones en el software. Si cambiamos un comportamiento, podemos ver si vamos a afectar y cómo, al resto de lógica del sistema incluso antes de integrar dicho cambio en el sistema de integración continua.

Los diferentes pasos que caracterizan el desarrollo TDD [2] son:

1. **Caso de uso:** El primer paso es definir el caso de uso que se va a probar y crear el test con dicho caso. Lógicamente este test fallará al crearlo, pues aún no hay nada de código.
2. **Código:** Una vez hallamos realizado el test, toca realizar el código. El código a realizar será el mínimo necesario para que funcione el test, si por ejemplo el código que estamos realizando necesita crear un método en otra clase, este código no se realizará, nos centraremos sólo en la clase que estamos probando.
3. **Refactorizar:** Para finalizar hay que revisar el código y simplificarlo a ser posible, en esta fase se eliminará el código repetido, tanto en el test cómo en la parte de código.

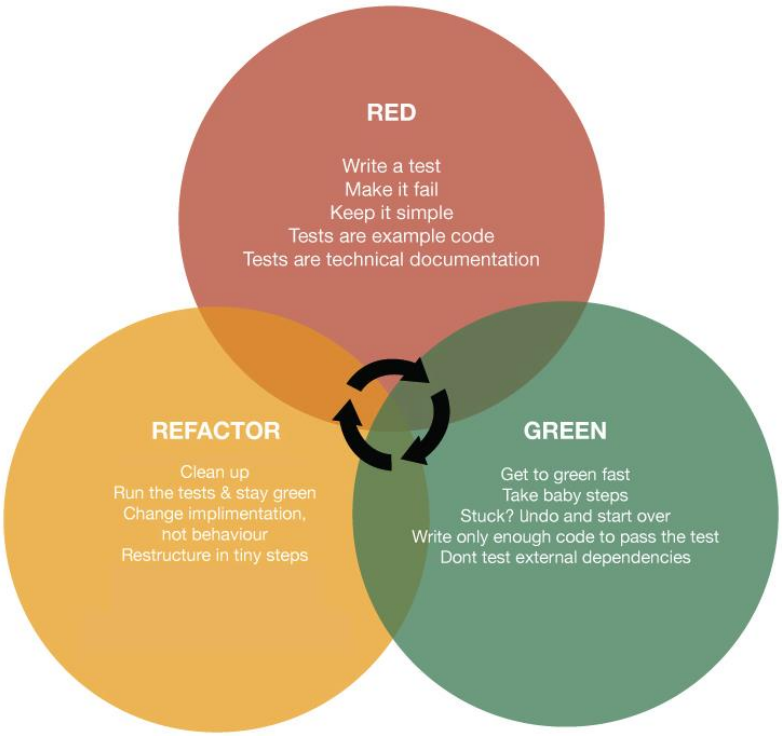


Figura 3-3 : Esquema TDD

Una vez hemos hecho las tres fases, se elige otro caso de uso y se vuelve a empezar, veremos cómo poco a poco todo el código de la clase se va rellenando y acabamos con una clase completa y bien organizada.

La Figura 3-3 [7] resume muy bien todo el proceso, y muestra porqué esta técnica se conoce también cómo Rojo-Verde-Rojo ya que es una técnica que desarrolla test para que fallen (rojo) para luego arreglarlos (Verde) y luego diseñar otro nuevo que falle (rojo).

3.3.3 SOLID

Es un acrónimo mnemotécnico que representa los 5 principios de la programación orientada a objetos [8]. Estos principios son:

1. **Single Responsibility Principle (SRP):** Cada clase debe tener una única funcionalidad, es decir, no se deben juntar varias funcionalidades en una sola clase.
2. **Open-Closed Principle (OCP):** Las clases deberían estar abiertas para ser extendidas, pero cerradas para ser modificadas.
3. **Liskov Substition Principle (LSP):** Un método debe funcionar igual de bien si recibe el objeto del tipo que espera, que si recibe uno de otro tipo que hereda del esperado.
4. **Interface Segregation Principle (ISP):** Un objeto no debe verse obligado a implementar una interfaz si no la utiliza.
5. **Dependency Inversion Principle (DIP):** Un módulo concreto A, no debe depender directamente de otro módulo concreto B, sino de una abstracción de B. Tal abstracción es una interfaz o una clase (que podría ser abstracta) que sirve de base para un conjunto de clases hijas.

Estos principios, los tendremos presentes a la hora de desarrollar. Dos acrónimos que resumen estas reglas son YAGNI (*You aren't gonna need it*) y KISS (*Keep it simple, Stupid*).

3.4 Estructura del proyecto

Para diseñar la estructura de la solución, nos hemos basado en los patrones MVC (Model View Controller) y MVP (Model View Presenter). Estos patrones sostienen que es necesario independizar la lógica de la aplicación, de la interfaz. Con esto en mente, desarrollaremos una solución (conjunto de proyectos de Visual Studio) con la siguiente estructura:

- **BuildFiles:** Archivos de nant build que usará TeamCity para generar las versiones del proyecto.
- **Dll:** Módulos externos usados por la solución, incluye los componentes Xamarin desarrollados por otros usuarios.
- **Servitrax.Android.App:** Proyecto principal de Android, es la aplicación Android con todas las actividades y recursos.
- **Servitrax.Android.BusinessLayer:** Librería pura de C# totalmente independiente del proyecto Android. Contiene toda la lógica de negocio de la aplicación.
- **Servitrax.Android.Core:** Librería Android que pone a disposición del resto de proyectos, métodos muy básicos como son el Logger y la gestión de fechas.
- **Servitrax.Android.DataAccess:** Librería Android que contiene los servicios para acceder a las base de datos y obtener información de las mismas. Todas las consultas a las bases de datos se harán a través de este servicio.
- **Servitrax.Android.DataAccess.Model:** Librería de C# que contiene las entidades de la base de datos, está separado del DataAccess para permitir a otros proyectos de la solución, utilizar objetos con la estructura de la base de datos sin necesidad de importar toda la librería de acceso. Esto será muy útil sobre todo para poder crear objetos de esos tipos de datos en los test.
- **Servitrax.Android.Platform:** Librería Android que contiene todos los servicios que dependen de Android (y que son específicos para la plataforma Android), en esta librería podemos encontrar servicios como sonido, conectividad, o

traducciones. Es decir, es un micro framework de servicios Android que serán utilizados en muchos puntos de la aplicación.

- **Servitrax.Android.Test.BusinessLayer:** Este proyecto es un proyecto dedicado a probar la lógica de la aplicación (BusinnessLayer) mediante test de NUnit. Será la base del desarrollo de la capa de negocio debido a que se programa siguiendo las técnicas indicadas en TDD.
- **Servitrax.Android.WebAccess:** Esta librería contiene la capa para comunicarse con el servidor, incluye los métodos para serializar, realizar llamadas web y los modelos para realizar y recibir las peticiones.

En general, nuestro sistema es un sistema separado por capas [4], cada capa sólo puede depender de capas más internas, y nunca puede una capa interna depender de una externa. En nuestro caso, la capa más interna sería la lógica de negocio junto con sus test, esta lógica estará "envuelta" en las interfaces de los servicios de dicha capa, aislando de esta forma las implementaciones de los servicios del uso de los mismos. A partir de ahí nos encontraríamos con el Core y la plataforma, que proveen al resto de proyectos de funciones básicas. Por último, y en la capa más externa iría la interfaz que utiliza las interfaces disponibles en la plataforma y en la capa de negocio.

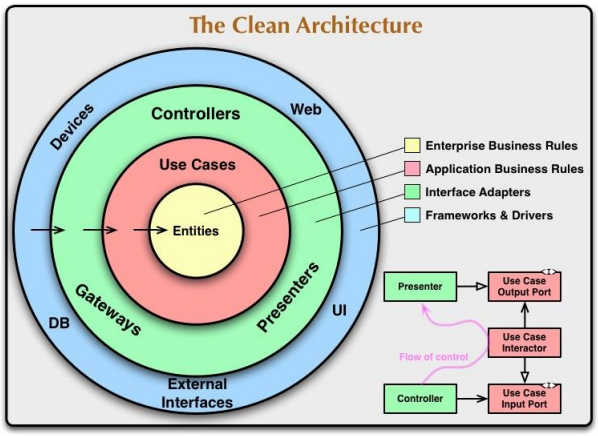


Figura 3-5 : Estructura diseño por capas

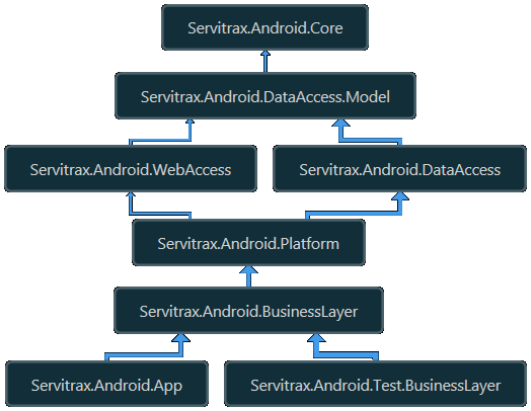


Figura 3-4 : Dependencias del proyecto

Podemos comprobar cómo la parte de lógica depende de la parte de Android, pero, utilizando interfaces, conseguiremos otorgarle de independencia. Esto facilita, el mantenimiento de la aplicación y la reutilización de código para nuevo proyecto. Además, esta organización facilita que si por ejemplo, se quisiese desarrollar para IOS (Sistema operativo de los IPhone) la parte de lógica de negocio seguiría valiendo ya que toda relación con Android se realiza a través de la interfaz de la plataforma. La plataforma nos indicará qué clases utilizar para los diferentes servicios siendo estos interfaces con los métodos necesarios. Esta estructura y manejo de los servicios viene aconsejado por los principios SOLID, ya que delegan la acción de los métodos en clases, pero las dependencias se realizan respecto a las interfaces. Además, esta estructura será muy fácil de probar porque podremos "mockear" los servicios para que devuelvan los datos esperados. Los *mocks* nos obligarán a que la lógica sea dependiente de la interfaz del servicio, y no de la implementación concreta, lo cual, unido al desarrollo TDD, nos obliga a que todos los servicios incluyan una interfaz y, cumplan los puntos 3 y 5 del SOLID.

3.5 Pruebas de concepto

Antes de comenzar el proyecto, una vez se tienen los requisitos y diseños finales, es necesario analizar y hacer pruebas de concepto de diferentes soluciones a problemas que se

prevén, por ejemplo, cómo realizar algunos elementos complejos de la interfaz o, determinar qué librerías se pueden utilizar y cuáles son las mejores.

3.5.1 Pruebas de concepto sobre SQLite

Para el desarrollo sobre SQLite, comprobamos que íbamos a necesitar un software que nos permitiese generar bases y modelos de datos de una forma cómoda, librerías que nos permitiesen cifrar la base de datos del dispositivo, y librerías que nos permitiesen realizar consultas sobre las bases de datos de una forma rápida y efectiva. De estas pruebas se sacó la conclusión de que, para la generación de la base de datos se iba a usar Entity Framework cambiando las plantillas de generación de bases de datos por unas hechas por nosotros que sirviesen para generar SQLite. Otra decisión que se tomó en estas pruebas, fue la de utilizar la librería de SqlLite-Net para la manipulación de las bases de datos, desde realizar consultas hasta modificar o eliminar datos.

3.5.2 Pruebas de concepto sobre la interfaz

Dentro de los diseños de la interfaz, existían varios elementos que nunca habíamos generado y de los que necesitábamos investigar las posibles formas de diseñarlos. Además, queríamos probar los gestos de deslizar sobre listas, sobre todo en aquellos casos en los que, según el diseño inicial, mostraban botones debajo de un elemento tras deslizar sobre él.

Por ejemplo, en la Figura 3-6 podemos ver un ejemplo de estos diseños. El diseño indicaba que tras deslizar sobre uno de los elementos de la lista, se mostrarían tres botones para añadir y eliminar servicios. Tras diseñarlo en un proyecto de prueba, se determinó que no era intuitivo (porque no se ve el elemento que estás modificando mientras lo haces) y que, no era cómodo de usar en el caso de que el usuario llevase guantes. Ante esta propuesta y al comprobar que no era viable, se contactó con el cliente y se le propuso otras opciones cómo la posibilidad de que el conjunto de acciones estuviese en un diálogo o la posibilidad de que el menú con las opciones se desplegase bajo el elemento en caso de que el usuario pulsase.

Otros elementos que se investigaron, pero que sí que llegarán a formar parte de la aplicación son los floating buttons (tal y como vienen recomendados en las guías de diseño de material design), el menú contextual, o las traducciones de forma dinámica.

Figura 3-6 : Diseño con Swipe en elemento

4 Desarrollo

4.1 Entorno de trabajo

Antes de comenzar a desarrollar es necesario preparar el entorno de trabajo, para ello instalaremos el software necesario (Visual Studio, Xamarin, Android SDK, …) y realizaremos los siguientes pasos:

- Crear repositorio SVN: Será donde se almacene el código y toda la documentación relacionada con el proyecto.
- Crear proyecto en Jira: Irá conectado al SVN y será donde el jefe de proyecto creará las tareas. De esta forma a la hora de subir código al repositorio, sólo podrá subir si hay un identificador de Jira en la descripción de los cambios realizados, y de esta forma, luego será más fácil encontrar la documentación relacionada con los cambios realizados.
- Crear proyecto en TeamCity: Cómo hemos visto, TeamCity nos servirá para generar versiones del proyecto verificando como sistema de integración continua que los componentes están correctamente integrados (pasan los test y compilan), por tanto, un paso clave, será configurarlo para que tenga la capacidad de compilar el código y realizar los test. Además, en nuestro caso, lo configuramos para que nada más subir un cambio al repositorio, compile el código, pase los test unitarios y de integración, y genere una versión para comprobar que no hay ningún problema en el código subido.
- Crear la solución en Visual Studio acorde a lo determinado en la fase de diseño y comenzar la primera actividad de nuestra aplicación o actualizarse a la última versión del repositorio si la solución ya ha sido creada.

4.2 Métodos de trabajo

Vamos a distinguir dos formas de trabajar dentro de nuestro proyecto:

- Trabajando en interfaz: Siguiendo los patrones MVC y MVP, se intentará tener el menor número de lógica posible en la clase encargada de la interfaz, en el caso de Android, en la actividad. Aquellas actividades que necesiten de métodos/procesos complejos para obtener algún dato, o procesos complejos a realizar tras una acción de un usuario, se llevarán a una clase aparte. De esta forma, facilitamos el mantenimiento de la aplicación porque cada actividad, contiene básicamente las funciones que componen su ciclo de vida, delegando en clases auxiliares los procesos complejos. Además, para unificar el diseño de la interfaz, existe un documento de la empresa que indica la forma correcta de nombrar los identificadores y variables.
- Trabajando sobre la capa de negocio: Sigue el modelo marcado por TDD, se diseña primero los test, y se va desarrollando el código según se va necesitando. Para cada grupo funcional de lógica de negocio, creamos uno o varios "servicios" independientes de la interfaz de usuario y del acceso a datos, que se comunican con el resto del sistema a través de BusinessObjects.

En ambos casos el diseño del código irá marcado por las reglas de StyleCop, esto hará que se consiga un código limpio y uniforme a lo largo de todo el proyecto. Tanto si se desarrolla interfaz cómo si se desarrolla capa de negocio, los pasos a seguir sin tener en cuenta la forma de hacer el código son los mismos:

1. Leer la descripción de la tarea que se va a realizar, la documentación y los requisitos relacionados.

2. Desarrollar el código, según las reglas marcadas por la parte del proyecto que vaya a verse afectada (TDD en el caso de la capa de negocio), comprobando antes, que las clases que vas a modificar, no causan conflicto con ningún compañero.
3. Revisar que el código no tiene fallos, probando a compilar de nuevo y ejecutando los test.
4. Comitear el código al servidor SVN, actualizando antes a la última versión para evitar subir conflictos si los hubiera. Para comitear será necesario indicar en el mensaje del *Commit* el identificador de la tarea en la que se está trabajando y una breve descripción de lo que se ha realizado, esto permitirá que, al ver el historial de cambios, se puede saber rápidamente las modificaciones que se han realizado.

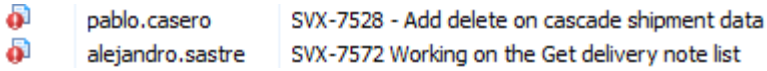


Figura 4-1 : Ejemplo historial proyecto

5. Escribir unas *Release Notes* donde se explique a nivel de cliente el desarrollo realizado en esa tarea y, opcionalmente si se ha desarrollado un componente reutilizable, se escribirá la documentación donde se explique cómo añadir y utilizar dicho componente. Conforme se acaba una iteración y se le entrega al cliente una versión, se entrega un documento donde se explica las diferentes mejoras que se han hecho, y en qué se ha invertido el tiempo durante dicha iteración, este documento, se generará a partir de la *Release Notes*. Además, en el caso de que se haya realizado un cambio en la interfaz o desarrollado una nueva pantalla, se incluirán capturas de pantalla. Esta información, será la que use el equipo de control de calidad para ejecutar sus test basados en herramientas de prueba automáticas.
6. Cerrar el desarrollo de la tarea en Jira para que otro desarrollador pueda revisarla más tarde.

4.3 Componentes desarrollados

Partiendo de la experiencia de otros proyectos Android, y buscando puntos en común dentro de los diseños de nuestra aplicación, nos dimos cuenta de varios componentes que podían ser reutilizados. Estos componentes, nos ahorrarán a la larga mucho tiempo, porque si bien hacer el componente genérico nos llevará más tiempo que el tiempo necesario para hacerlo de un solo uso, cada vez que tengamos que reutilizarlo nos costará mucho menos. Un ejemplo de esto es el menú contextual, es un componente que desarrollarlo de forma genérica nos llevó 5 días mientras que generarlo para la primera pantalla, podíamos haberlo acabado en tan sólo 2, pero, teniendo en cuenta que hasta 50 pantallas llevan un menú contextual, desarrollarlo de forma genérica nos supondrá un gran ahorro de tiempo reduciendo el desarrollo de cada menú a un par de horas (tiempo de pruebas incluido).

4.3.1 Menú Contextual y cabecera

Tanto el menú contextual como la cabecera, están presentes en toda la aplicación una vez el usuario ha iniciado sesión en la aplicación. Por ello, los hemos desarrollado de forma que sean fáciles de incluir en todas las actividades, e irán siempre unidos, porque uno de los botones de la cabecera sirve para mostrar el menú contextual. A la hora de hacer el componente, tendremos que analizar que partes van a ser comunes en todos los casos y qué partes en cambio van a ir cambiando según el uso.

Figura 4-2 : Cabecera de la aplicación

Por ejemplo, cuando se esté en el menú principal, en el menú contextual sólo saldrá la opción de salir de la aplicación o acceder a los ajustes, en cambio, cuando se está viendo un pedido, el menú incluirá opciones para ordenar, sincronizar, subir cambios al servidor y otras opciones relacionadas con el pedido.
Para incluirlos en una actividad, será necesario incluirlo en el layout de la actividad, y será necesario incluirlo en el código de la actividad propiamente.

- **Layout:** Un menú contextual, corresponde con el elemento DrawerLayout de Android. Este elemento, tiene que incluir primero el contenido de la actividad y luego el del menú, siempre en ese orden y entre las etiquetas de apertura y cierre del layout. Debido a que necesita esta estructura determinada, no será un elemento fácil de incluir. Para incluir en una actividad el menú contextual con su cabecera, será necesario que se use una estructura determinada para la actividad.
  Esta estructura es la que se aprecia en la Tabla 4-1 donde podemos comprobar cómo se incluyen dos layouts importantes.
  Por un lado, dentro de la región destinada a definir la interfaz de la actividad, incluimos el *Header* (cabecera) que incluirá el botón para abrir el menú. Por otro lado, incluimos el layout del menú contextual que contiene un grid para disponer los elementos del menú y los botones de salir y ajustes. Además, se diseñará otro layout para el menú contextual sin grid, que se usará en aquellos casos en los que el grid no contenga ningún elemento. Por tanto, para estos casos, bastará con incluir el layout *Contextual_Menu_Without_Grid* en lugar del normal.

Tabla 4-1 : Esquema layout con Contextual Menu

```
<LinearLayout>
    <android.support.v4.widget.DrawerLayout
        android:id="@+id/Contextual_Menu_DrawerLayout">
    <!-- Contenido de la actividad -->
        <LinearLayout>
            <include
                layout="@layout/Header" />
            <LinearLayout>
            <!-- Contenido -->
            </LinearLayout>
        </LinearLayout>
    <!-- El menú contextual -->
        <include
            layout="@layout/Contextual_Menu" />
    </android.support.v4.widget.DrawerLayout>
</LinearLayout>
```

- **Código de la actividad:** Tanto el menú contextual cómo el header, tienen elementos y acciones que tienen que ser declarados siempre que se utilizan. Entre otros, encontramos la asignación del evento de atrás al botón de la cabecera, la acción de abrir el menú al pulsar el botón correspondiente en la cabecera o la acción del botón de ajustes del menú contextual. Para todas estas acciones comunes, y evitar repetir código, desarrollamos un helper que realizase todas estas acciones y que, además, incluyese métodos para la gestión de los elementos del Grid. Entre estos métodos, cabe destacar dos:
  - **FillMenu:** Este método recibe un diccionario con las imágenes y las acciones correspondientes a cada imagen y crea el grid para que contenga estos elementos con sus acciones correspondientes ya asignadas.
  - **ChangeState:** Este método nos permite cambiar el estado (habilitar/deshablitar) de los elementos

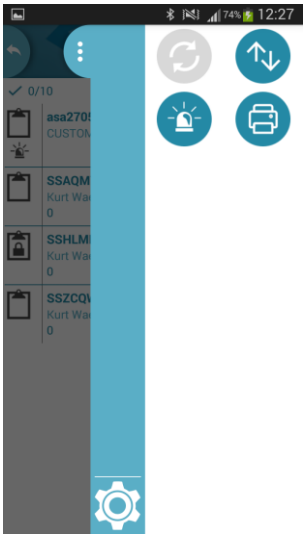


Figura 4-3 : Ejemplo menú contextual

del grid. Este método nos será muy útil debido a que muchos requisitos incluyen la desactivación de un elemento del menú si no se cumplen unas condiciones.

4.3.2 Diálogos

Una parte fundamental de toda aplicación Android, son los diálogos, que sirven para mostrar errores, pedir confirmación de acciones … En especial, tendrán gran importancia los diálogos de error y confirmación que serán usados a lo largo de toda la aplicación. Por ello, los desarrollaremos con suficiente flexibilidad cómo para que nos sirvan para todas las situaciones y de forma que no tengan una gran complejidad de uso. Para facilitar la generación de diálogos, aparte de la clase que conforma el diálogo con su layout, generaremos una clase auxiliar que tendrá métodos públicos para construir diálogos de una forma más sencilla. Entre estos métodos encontraremos métodos simples que van directamente enfocados a crear diálogos específicos y otros métodos enfocados a dar libertad al usuario para crear los diálogos como quiera.
Los métodos específicos los fuimos creamos cuando nos dimos cuenta que creábamos diálogos con mensajes que tenían muchas partes en común, por lo que decidimos facilitar su creación y reducir de esta manera el código repetido. Además, estos métodos nos permitirán que las acciones no tengan que estar embebidas dentro de los diálogos, lo que nos facilita el diseño al poder sacar dichas acciones a otras clases.
Un ejemplo de estos métodos es el usado para crear mensajes de error:

```
public static MessageDialog CreateErrorMessageDialog(Activity activity, string
 content, EventHandler buttonHandler)
```

Que nos devuelve el diálogo creado con el contenido y el evento que se hayan pasado por parámetros con la estructura de un mensaje de error, y como vimos que, en la mayoría de los casos lo único que queríamos era quitar el diálogo al hacer una acción, añadimos la opción de que si no se le pasa ninguna acción (`EventHandler == null`) se le asigna la acción de cerrar el diálogo (*dismiss*).
Como queremos que nuestros diálogos tengan una apariencia personalizada por nosotros, hemos tenido que desarrollar una implementación propia de diálogo (*MessageDialog*) extendiendo de DialogFragment, esto nos proporcionará la base con la que desarrollar nuestro diálogo. A partir de aquí, haremos *override* de los métodos *OnCreateView* y *OnViewCreated* que serán los métodos que usará Android para crear los diálogos. Con el primero le indicaremos a Android que layout inflar (qué xml usar para crear la vista) y con

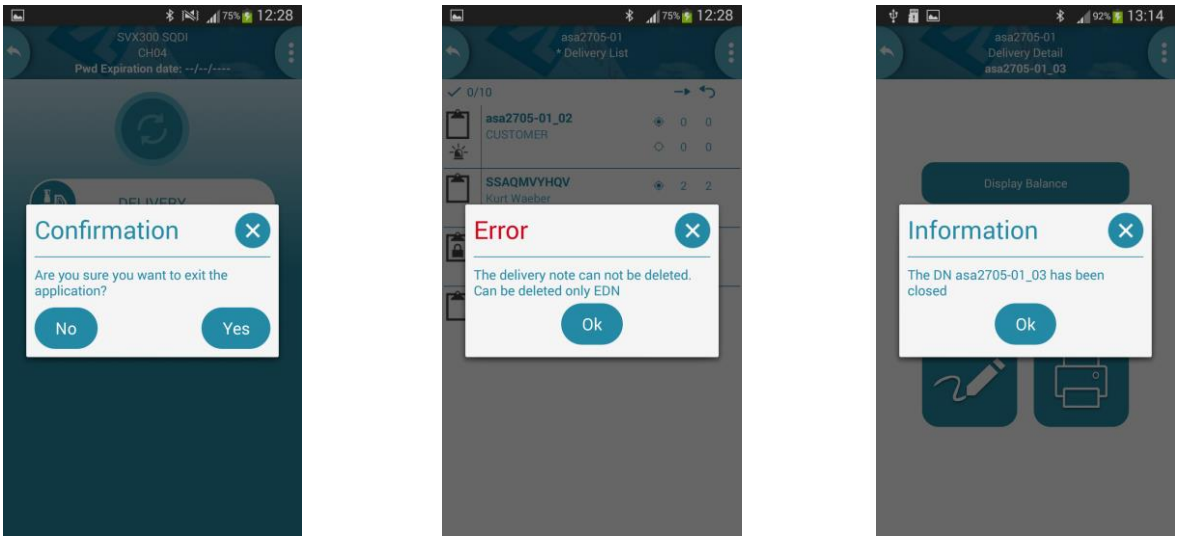
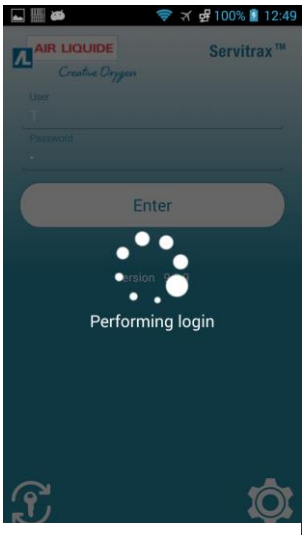


Figura 4-4 : Ejemplos diálogos

el segundo, rellenaremos la vista con la información que tengamos y asignaremos eventos a los botones. Cómo no hay la misma cantidad de botones en un diálogo de confirmación que en uno de mensaje, y queremos que el diálogo nos sirva para realizar los dos tipos de diálogo, crearemos dos métodos diferentes para rellenar la vista y usaremos uno u otro según el tipo de diálogo que estemos creando.
Ambos tipos de diálogos usarán el mismo layout, la diferencia radica en que el de mensaje (ya sea de error o de ok) no mostrará el segundo botón, sólo mostrará uno de los dos con el nombre correspondiente.
Antes de darlos por finalizados, vimos en el documento de requisitos, que el cliente indicaba que, si el diálogo era de error, el diálogo tenía que ir acompañado de un sonido a máximo volumen. Por ello, y utilizando un servicio creado por nosotros para reproducir sonido, decidimos crear un método público para que se usase en lugar del de por defecto, para mostrar el diálogo. El método comprobará si el diálogo es de error y si es así, llamará al servicio encargado de reproducir el sonido antes de mostrar el diálogo. Además, para mejorar la usabilidad, prevemos meter además del sonido de error una vibración, esta estructura de implementación con clase auxiliar nos permitirá hacerlo tocando en sólo un punto.

4.3.3 Custom Progress Dialog

Este diálogo es el sustituto de un *ProgressDialog* de Android, ambos son diálogos destinados a ser mostrados durante un proceso de larga duración para indicar al usuario que se está trabajando en la acción solicitada, o indicarle que hay un trabajo en proceso. Además, evitan que el usuario continúe usando la aplicación antes de que el proceso haya finalizado. En nuestro caso, el diálogo se encargará de oscurecer la pantalla y mostrar un icono girando con la descripción del proceso que se está llevando a cabo. Este planteamiento generará varios problemas:



- Necesitamos que el proceso se realice en segundo plano, sin interrumpir la animación.
- Necesitamos ser capaces de comprobar si el proceso ha tenido alguna excepción o si por el contrario se ha desarrollado correctamente.
- Necesitamos ser capaces de indicar el paso siguiente una vez acabe el diálogo con éxito.
- Necesitamos poder actualizar el texto del diálogo según avanza el proceso.
- Necesitamos que el diálogo tenga una animación que empiece cuando empiece el diálogo.
- Queremos que sea sencillo de usar y evitar la repetición de código lo máximo posible.

Figura 4-5 : Process Dialog

Para permitir todo esto, diseñaremos la clase (*BackgroundProgress*), encargada de manejar el proceso y lanzar/ocultar el diálogo, una clase (*CustomProgressDialog*) que herede de *ProgressDialog* con los métodos propios para manejar el diálogo, y una clase (*BackgroundWorker*) que almacene, el método a ejecutar durante la acción, el método a ejecutar cuando finalice, el método a ejecutar cuando el proceso ejecutado lance una excepción y un listener desarrollado para actualizar el texto del diálogo. Esta clase no realizará ninguna funcionalidad más que notificar al diálogo del mensaje inicial.
El diálogo, es una extensión de ProgressDialog, en esta extensión, haremos que el diálogo muestre la interfaz que nosotros queremos (fondo transparente e icono que gira), haremos

que los métodos para mostrar y ocultar el diálogo den comienzo/paren la animación, y haremos un método para poder cambiar desde el proceso el texto que se muestra.
La clase que maneja el proceso será desde donde se inicializará y controlará todo el proceso de mostrar el diálogo, empezar proceso, notificar de los cambios en el proceso al diálogo, realizar la acción de cuando acaba el diálogo, … Con este objetivo, inicializaremos el diálogo utilizando el siguiente método:

```
public void Init(Activity act, Action<BackgroundWorker> processAction, Action<Exception>
                exceptionAction, Action endAction, string dialogMessage)
```

Con esta información, inicializaremos el *BackgroundWorker* con los procesos y guardaremos este *worker* en la clase. El listener para modificar el texto del diálogo será un método dentro del Progress que llama al método correspondiente del diálogo.
Una vez tenemos el worker inicializado, esperamos a la llamada del método Show para empezar todo el proceso, el cual no se podrá parar una vez se le dé comienzo. Una vez se llama al método Show, crearemos un *CustomProgressDialog* con el mensaje inicial y comenzaremos una *Task* con el método RunProcess indicando que al acabar se ejecute OnEndProcess.

- **RunProcess:** Consiste en una llamada al método guardado en el worker enviando cómo parámetros el propio worker para que, desde dentro del método declarado y fuera del objeto, se pueda llamar al listener encargado de cambiar el texto del diálogo. Además, se fuerza una duración mínima predefinida para asegurarnos que siempre se llega a mostrar el diálogo.
- **OnEndProcess:** En este método lo primero que se hace es quitar el diálogo de progreso (pues ya ha acabado) y comprobar si la tarea ha finalizado correctamente. En caso, afirmativo, se llama al método OnFinish guardado en el worker y en caso negativo se llama al método OnException del worker y se le pasa la excepción que ha hecho que el proceso finalice con error.

Un ejemplo de cómo se usaría este diálogo sería el siguiente:

```
// Creates a progress dialog with the backgroundProcess1and2 as process.
BackgroundProgress backgroundProgress = new BackgroundProgress();
string message = “Processing 1...”
backgroundProgress.Init(this.activity, this.BackgroundProcess1and2,
                        this.OnException,  this.OnFinish, message);
backgroundProgress.Show();
```

```
public void BackgroundProcess1and2(BackgroundWorker worker){
// Processing 1
worker. UpdateProgress(“Processing 2”);
// Processing 2
}
```

```
public void OnException(Exception exception) { … }
```

```
public void OnFinish(){ … }
```

Tabla 4-2 : Ejemplo de uso Custom Dialog Progress

Podemos comprobar cómo el código de uso del diálogo queda claro, limpio y evita tener que repetir código cada vez que queremos un diálogo de este tipo.

4.3.4 Loggeador (LoggerService)

A lo largo de toda nuestra aplicación, será necesario llevar un registro de todas las acciones que realiza la aplicación con el objetivo de que, si nos notifican una incidencia, sea fácil de averiguar el proceso seguido por el usuario hasta tener el error, y así poder replicarlo luego y averiguar la causa. Para ello, conforme el usuario realice acciones o entre en pantallas, iremos escribiendo en el log la acción realizada por medio del servicio. Este servicio nos permitirá escribir los 4 tipos de logs explicados en la Tabla 4-3 según la funcionalidad realizada, cada uno de estos logs irán en archivos por separado.

Loggeador	Archivo	Tipo
AppLogger	`servitrax.log`	Principales procesos de la aplicación.
UiLogger	`servitrax_ui.log`	Interfaz
DalLogger	`servitrax_dal.log`	Acceso a datos
WebLogger	`Servitrax_web.log`	Acceso a la red

Tabla 4-3 : Tipos de Log

Además, distinguiremos también 4 tipos de nivel de log para distinguir los logs de errores graves, de los log de información en los que se guarda por ejemplo, en qué pantalla acaba de entrar el usuario. Los niveles de log ordenados de menor a más graves quedarían de la siguiente forma:

DEBUG < INFO < WARNING < ERROR

Por tanto, el servicio de log, no sólo nos tiene que permitir loggear según el tipo, sino también según el nivel de error que queramos. Además, dentro de los ajustes de la aplicación se le permitirá al usuario elegir el nivel de log que desea utilizar, de esta forma si el usuario elige el tipo de error, sólo se loggearán los mensajes de error mientras que, si elige Info se loggearán todos los mensajes menos los de Debug. Dentro de los ajustes también se podrá especificar qué logs se desean generar no siendo posible la desactivación del log de la aplicación (log principal).
Al ser un log un archivo en el que se escribe una gran cantidad de información, se decidió limitar el tamaño del log a 15 archivos de log de cada tipo, con máximo de 2MB cada uno, de forma que, si el archivo servitrax_0.log alcanza los 2MB, este archivo se renombrará a servitrax_1.log y se seguirá escribiendo en un nuevo servitrax_0.log, así hasta servitrax_14.log. Una vez alcanzado el límite, se irán eliminando los registros del log más antiguo (se elimina el 14, el 13 se renombre al 14, … y se sigue escribiendo en el 0). Esta técnica de log es conocida como *Rolling file appender* y nos permite tener un log bastante completo de lo que ha sucedido sin ocupar un espacio que crece sin límite.
En cada línea de log se guarda, además del mensaje, la clase que ha creado el log, el método, el nivel de error, y la fecha. Esta información se guardará con la siguiente estructura:

Fecha | Nivel de Log | Clase :: Método | Mensaje

De forma que un mensaje de log quedaría:

```
23-05-2016 10:04:18.165 | INFO | md5a396316f335dae6b14724037fa888d82.ServitraxApp :: n_onCreate | Change the device connectivity True. Is connected by WIFI or ethernet True. Is connected by GPRS False
```

A la hora de loggear un mensaje, y siguiendo el mismo esquema que el resto de servicios genéricos, nos interesa que sea fácil de usar por parte del desarrollador, por ello, tendremos

un servicio en la plataforma y desde él, podremos acceder a los diferentes tipos de loggers. Un ejemplo de cómo se loggearía un diálogo en el log sería:

```
ILoggerService myLogger = AndroidPlatform.Instance.LogService;
myLogger.UiLogger.LogDebug("Lanzando diálogo de proceso");
```

4.3.5 Servicio de sonido

A lo largo de nuestra aplicación, hay varios casos en los que tendremos que acompañar un mensaje o acción con un sonido a modo de alerta para el usuario. Debido a que, gran parte de los usuarios que utilizan el terminal, trabajan en condiciones con mucho ruido, el cliente estableció como requisito de la aplicación, que los sonidos sonasen al máximo volumen sin importar la configuración del usuario en el terminal. Por tanto, para facilitar la reproducción de los sonidos, decidimos crear un servicio que se encargase de esta tarea. Añadimos el servicio a la plataforma, que, como el resto de servicios, no se añadirá la implementación del servicio, sino que se añadirá la interfaz del mismo y este servicio tendrá un método por cada sonido que se va permitir reproducir, cada sonido corresponderá a un archivo presente en la Aplicación y, desde un archivo de configuración se indicará que archivos sirven para qué sonidos. Esto facilitará el cambio de estos sonidos por otros nuevos sin necesidad de cambiar ninguna línea de código o de sacar una nueva versión. Un ejemplo de una línea donde indicamos cual es el archivo de error sería:

```
Sound_Error = Error.aac
```

Gracias a esta propiedad, luego en cada método podremos acceder a esta configuración y encontrar el archivo indicado para cada método.

4.3.6 Comprobar Conectividad

A lo largo de nuestra aplicación, existen múltiples elementos que su disponibilidad está totalmente relacionada con el estado de la conexión, esto se debe a que dependen de la conexión para funcionar, por lo que sus botones serán deshabilitados cuando no tenga conexión el terminal. Este tipo de relación directa (conexión-habilitación) se puede hacer creando en todas las actividades un listener que salte cada vez que cambie la conexión, o indicar a una clase, qué elementos son los que queremos que cambien de estado conforme cambia la conexión. Este segundo sistema será el que usemos porque reduce la comprobación de conexión a un solo punto.

4.3.7 Traducciones

Android ofrece soporte nativo para gestionar las traducciones de diferentes idiomas, para ello, el desarrollador sólo tiene que añadir las cadenas de caracteres en el archivo strings.xml de la carpeta del idioma que se quiera (values-[códigoDelIdioma]). Este sistema, seleccionará las cadenas de caracteres que encuentre en la carpeta del idioma del dispositivo y, si alguna cadena no la encuentra en la carpeta del idioma, irá a la carpeta por defecto. Este sistema funciona muy bien, pero tiene una desventaja que hizo que decidiéramos no usarlo, esta desventaja es que, si se quiere actualizar una traducción, es necesario generar un paquete nuevo con todo lo que eso conlleva (reléase candidate, preproducción, cliente, despliegue, …). Por ello, nosotros necesitábamos un sistema que nos permitiese a nosotros y al cliente, actualizar las traducciones sin necesidad de crear y desplegar nuevas versiones.

La solución fue utilizar un Excel con las traducciones, y un proyecto de C# encargado de pasar las traducciones del Excel a una base de datos sqlite. Este proceso se realiza automáticamente por el servidor de integración continua mediante la ejecución del proyecto de C#, con ello que obtendremos la base de datos que utilizaremos para actualizar las bases de datos de los usuarios cuando inicien sesión, sin necesidad de generar una nueva versión de la aplicación.

Desde el punto de vista del desarrollador, la obtención de las traducciones las haremos a través del servicio de traducciones presente en la plataforma. Este servicio, publicará métodos públicos para la obtención de cadenas de caracteres traducidas en la base de datos. Destacaremos aquí dos tipos de traducciones:

- Traducciones de partes de la aplicación: Son las traducciones de los elementos de la interfaz, las traducciones de estos elementos estarán en la base de datos y en los archivos por defecto de Android. De esta forma, nuestro método para obtener traducciones usará el identificador de la cadena que use Android para localizar la cadena en la base de datos y, en caso de no encontrarlo en la base de datos, cogerlo del xml presente en la aplicación, pero añadiéndole un * delante para indicar que dicha cadena no ha sido obtenida de la base de datos, con la finalidad de detectar las traducciones que faltan por adelantado.
- Traducciones comunes entre sistemas: Son las traducciones que se comparten con el resto de elementos de la estructura de Servitrax (Servidor, *Rich Client*, …). Estas traducciones se buscarán directamente en la base de datos ya que no formarán parte de la aplicación. Para obtenerlas, en vez de usar el identificador de Android, se usarán cadenas de caracteres que representen a cada traducción. Estas traducciones no se obtendrán de la base de datos generada por el Excel, sino que las proporcionará el servidor.

Un ejemplo de uso de este sistema de traducciones que nos traduciría el título de un diálogo es:

```
this.translationService = AndroidPlatform.Instance.TranslationService;
this.translationService.GetString(Resource.String. Dialog_Tittle);
```

En el caso de las traducciones comunes, la forma de obtenerlas sería la misma, pero usando una cadena de caracteres que identifique a la traducción que buscamos.
Las traducciones obtenidas, serán del idioma que tenga especificado el usuario en los ajustes, además, cuando el usuario realiza login se actualizará este ajuste y el idioma con el idioma que tenga el usuario introducido en su perfil.

4.3.8 CurrentTransaction

Servicio muy simple, presente en la plataforma que se utiliza para ir guardando variables útiles sobre el estado actual de la transacción en curso que realiza el usuario. En este servicio guardamos información como el usuario actual que ha realizado login, el viaje seleccionado para distribuir, el pedido que se va a entregar o el activo que vamos a distribuir. Esto nos servirá para facilitar el acceso a esta información importante y para facilitar la comunicación entre las actividades, quitando la necesidad de pasar variables como argumentos.

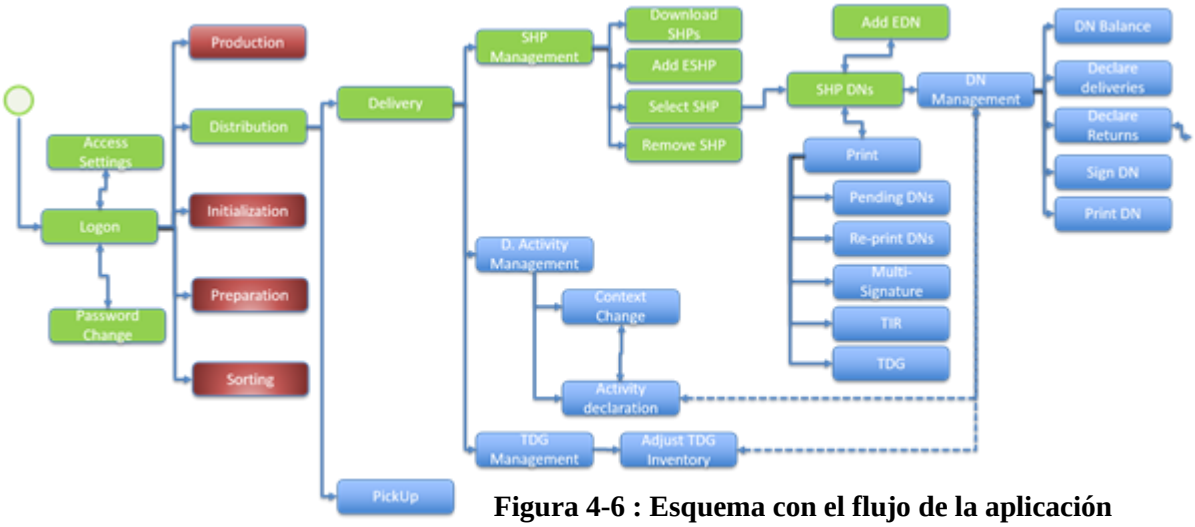


Figura 4-6 : Esquema con el flujo de la aplicación

4.4 Desarrollo de la aplicación

La primera versión de nuestra aplicación cubre la funcionalidad correspondiente con la distribución de las botellas. En la Figura 4-6, podemos ver las diferentes pantallas y funcionalidades que están incluidas dentro de esta versión (azul), las partes pertenecientes a futuras versiones (rojo) y las partes que ya han sido desarrolladas (verde).
Cada una de las pantallas que se han desarrollado, siguen las guías de diseño marcadas por el cliente y hacen uso de los componentes desarrollados, los cuales no sólo se usarán en las pantallas de esta versión, sino que también serán usados por el resto de versiones.

4.4.1 Splash Screen

Una splash screen, es una pantalla que se muestra nada más empezar la aplicación. Esta pantalla no tendrá ninguna lógica, es una pantalla que se muestra durante el tiempo de carga de la aplicación para que el usuario sepa que la aplicación se está cargando. En nuestro caso, además estará personalizada según el entorno sobre el que se trabaje (Canada, Europa, MyGasPartner).
Esta pantalla, también se usará cómo medio para salir de la aplicación, aunque no llegue a mostrarse al salir. Esto se debe a que, para salir en una aplicación de Android, es necesario, lanzar una actividad (la pantalla de splash en este caso) y limpiar la cola de actividades, para que cuando se cierre la actividad lanzada, no haya ninguna actividad de la aplicación detrás y se cierre por tanto la aplicación. Por ello, lo que haremos para salir, será limpiar la cola de actividades y llamar a la actividad del Splash con una variable indicándole qué, en vez de mostrarse, se cierre.

Figura 4-7 : Splash Screen

4.4.2 Ajustes

Una parte importante de nuestra aplicación serán los ajustes, se tendrá acceso a la pantalla de ajustes desde cualquier otra pantalla, pero si no se ha iniciado sesión, el acceso estará protegido mediante contraseña para evitar que usuarios sin permiso cambien la configuración de la aplicación.
Dentro de la pantalla de ajustes, tendremos la posibilidad de:

- Cambiar idioma
- Eliminar los datos del dispositivo, esto incluye bases de datos y configuraciones.
- Configurar impresoras
- Configuración avanzada
    - Cambiar la dirección, puerto y protocolo del servidor
    - Gestionar los logs activos y su nivel de criticidad
    - Cambiar el entorno de la aplicación

Figura 4-8 : Pantalla de ajustes

4.4.3 Bases de datos

Como hemos ido comentando, las aplicaciones Android usan SQLite como gestor de base de datos por lo que, todas nuestras bases de datos usarán este sistema. Dentro de nuestra aplicación tendremos 3 bases de datos:

- Translations: Aquí almacenaremos las traducciones de la aplicación para todos los idiomas soportados que actualmente son más de 10.
- Distribution: Almacenaremos los datos de distribución, los pedidos y envíos, los productos escaneados y toda la información relacionada con la parte de distribución.
- Aunthentication: Aquí almacenaremos los datos de login, esto nos permitirá hacer login incluso sin conexión a internet.

Tendremos las separadas bases de datos separadas por funcionalidad para poder realizar un mantenimiento de las mismas por separado, permitiéndonos cambiar el modelo de traducciones sin necesidad de rehacer todas las tablas de distribución. En concreto, cada fase nueva del ciclo de vida de las botellas que realicemos en futuras versiones, la iremos desarrollando en bases de datos separadas permitiéndonos así mantener una independencia entre todos las partes de la aplicación y manteniendo bases de datos más manejables.

4.4.4 Pantalla de pedidos

Centrándonos en el objetivo de la aplicación que es el de conocer las fases por las que va pasando un producto a lo largo de su ciclo de vida, podemos poner como ejemplo, la pantalla de pedidos donde el usuario una vez ha elegido un viaje, podrá mostrar los pedidos que tenga ese viaje y ver de un vistazo toda la información de cada uno.

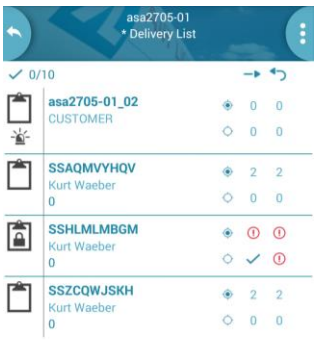


Figura 4-9 : Pantalla de pedidos

Esta pantalla, está formada por la cabecera y el menú contextual que tendrá las funciones específicas de esta pantalla (ordenar lista, actualizar los pedidos, imprimir el viaje, y crear un pedido de emergencia) y por la lista de pedidos. En esta lista cada uno de los elementos corresponderá con un pedido a un cliente, y, el último elemento seleccionado, se quedará marcado para que el usuario sepa el último pedido que modificó. En cada uno de ellos, se podrá ver toda la información relacionada con el pedido de forma simplificada. Esta información incluye, identificador del pedido, nombre del cliente, código postal y el número de productos a entregar separados por categoría (productos identificados y productos no identificados) y diferenciados entre entregas ( ) y recogidas ( ). Por último, se indicará también el estado de los pedidos. Además, cuando un pedido se haya modificado o cerrado, los números de productos se sustituirán por iconos indicando si se han entregado/recogido la cantidad correcta de productos.

4.5 Desarrollo mediante test

Siguiendo el esquema definido por TDD conseguiremos obtener el código de los servicios de una forma sencilla y ordenada. Veamos pues, qué pasos se dan en nuestra aplicación para conseguirlo:

1. Creación del servicio a desarrollar en la capa de negocio utilizando la plantilla de clases existente.
2. Creación de la clase de test que se encargará de probar la clase.
3. Creación de la inicialización del test, el *SetUp* para construir el punto de partida de los test, y el *TearDown* que se encargará de limpiar los cambios realizados.

4. Crear el test del método que corresponde con un caso de uso que vayamos a desarrollar, esta tarea incluye la creación del método en la clase, que de momento método estará vacío.
5. Diseñar el caso de uso en el test. El test diseñará las entradas al método y comprobará mediante aserciones, si la salida es la esperada. Cómo aún no hemos diseñado nada en la clase, este test fallará. (Rojo)
6. Desarrollar el método para que realice la acción que espera el test. Si este método necesita crear otros métodos en otras clases, estas no se desarrollarán, debido a que se prueba una clase cada vez. Los métodos extra se realizarán una vez hayamos acabado con la clase actual. Veremos cómo conseguiremos mediante Mocks que el método desarrollado funcione, aunque no estén desarrollados los métodos auxiliares.
7. Comprobar que el test funciona (verde) y continuar con otros casos de uso.

Como vemos, con el desarrollo de los test, realizando cada caso de uso, conseguimos desarrollar la capa de negocio al completo.

4.5.1 Uso de Mocks

Los Mocks, son clases que simulan otros objetos. Nos servirán para representar el funcionamiento de clases auxiliares a la que estamos probando [5]. El mock de una clase, no contiene nada de funcionalidad ni de lógica. Por ejemplo, el mock de un método que devuelve un valor, será un método que devuelva un atributo público de la clase del mock.

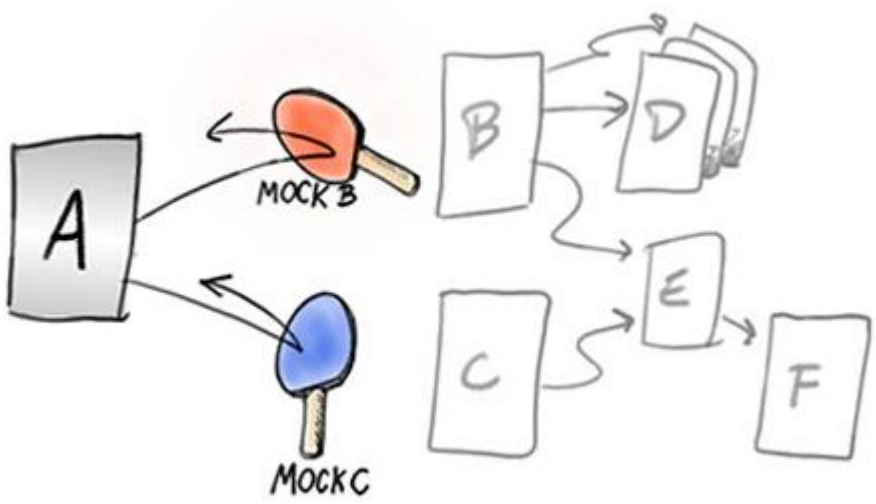


Figura 4-10 : Esquema Mocks

Nosotros los usaremos, por tanto, desde el test, para poder sustituir la clase que implementa dicho método por el mock, y forzar que dicho método devuelva el valor deseado (en el ejemplo, asignando a esta variable previamente un valor).

Para tener la posibilidad de sustituir la clase auxiliar por otra mockeada, será fundamental que dicha clase tenga una interfaz de forma que, la clase a probar, tenga una interfaz y en el test, esa interfaz la implemente el mock y luego en la aplicación de verdad, se utilice la implementación completa. Es una parte fundamental del TDD y contribuye a desarrollar software que respeta los principios de SOLID.

5 Integración, pruebas y resultados

Dentro de cada iteración, la última semana está normalmente reservada para pruebas, además, cada tarea que ha sido realizada debe ser revisada y se debe preparar una versión para entregar al grupo de control de calidad para que éstos lo entreguen, en caso de ser una *Release Candidate*, al cliente en un entorno de preproducción.

5.1 Revisión de tareas

Cuando un desarrollador finaliza una tarea, esta no pasa al estado de "Finalizado" sino que pasa al estado de "To Review" para que otra persona del equipo pueda revisar la tarea. En esta fase, el probador (desarrollador del equipo que le ha tocado revisar esa tarea), revisa que la tarea se ha realizado correctamente y que no hay nada sin realizar. El modo de probar que usamos, viene indicado por los puntos de vista desde los que se analiza el trabajo realizado:

- **Cliente:** Se comprueba que hay *Release Notes*, que lo descrito en la tarea se cumple, y que se explica lo que se ha hecho de manera sencilla y con capturas de pantalla.
- **Desarrollador:** Se comprueba que el código desarrollado cumple los requisitos funcionales, se comprueban los comentarios y el Stylecop, se comprueba que el código tiene un diseño y código limpio y si se reutilizan partes ya hechas. También se asegurará, de que las particularidades complejas u ocultas estén bien explicadas.
- **Tester:** Se buscan casos de prueba que no se le ocurrieron al desarrollador y se documentan y prueban, adjuntado archivos y resumen de los resultados.
- **Usuario:** Se comprueba la interfaz, que los mensajes sean correctos (indican lo que ocurre en la aplicación) y que están bien traducidos, se comprueban los casos de uso más utilizados, y se comprueba que la información que se escribe en el log es correcta y explica correctamente las acciones realizadas en el terminal.

Tenemos comprobado que, si se analizan las tareas siguiendo estos perfiles, conseguimos una corrección completa.

Al finalizar la revisión, si se comprueba que hay fallos o partes que faltan, existe la posibilidad de rechazar la tarea añadiendo un comentario, al hacerlo, Jira notifica inmediatamente al usuario de que tiene que arreglar la tarea. En caso de no haber fallos, el probador podrá cerrar la tarea para que quede en estado de finalizada.

5.2 Revisión de la versión

Antes de entregar una versión al equipo de control de calidad, se prueba la aplicación en profundidad en modo *Release* (no desarrollo) con un terminal real. Esta versión, se generará desde el TeamCity desde donde se obtendrá el apk con el que se realizarán las pruebas.

Estas pruebas se centrarán en probar la aplicación como conjunto, verificando los casos de uso, navegando entre pantallas con distintas condiciones como cambios en la conectividad, verificando que la información se muestra e integra correctamente, etc.

5.3 Entrega de versión

Una vez tenemos la aplicación probada y todas las tareas de la versión finalizadas, podremos proceder a entregar al control de calidad la aplicación. Esta entrega estará formada por:

- Paquetes desarrollados.
- Herramientas resultantes de pruebas de concepto y que consideramos pueden ser útiles para el cliente.
- Release notes: Indicando las novedades y los flujos de pantallas.
- Guías de instalación
- Scripts SQL para la regularización de la Base de datos central.

Todo ello, se entregará a través de un servidor FTP al que el equipo de control de calidad tendrá acceso. Estas entregas incluirán tanto nuestra aplicación, cómo el *Wrapper*.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

A la hora de enfrentarse a grandes proyectos, es igual o incluso más importante la fase de diseño que la de desarrollo. Una mala planificación o un mal diseño, llevarán al proyecto al fracaso porque, aunque se consiga finalizar el proyecto, el código se volverá enseguida difícil de mantener, empezará a tener errores, ... En cambio, un buen diseño y una buena planificación, organizarán al equipo correctamente consiguiendo un desarrollo eficiente, y al definir unas técnicas de trabajo, se obtendrá un código limpio y fácil de mantener, gracias a los comentarios, la documentación y a seguir unos patrones de diseño determinados. Por ello, mi etapa desarrollando este proyecto me ha sido muy útil, porque he aprendido a usar las metodologías de trabajo de forma eficiente poniéndolas en práctica, a utilizar patrones y principios de diseño que dirigen la implementación del código hacia un desarrollo limpio, y además he aprendido indirectamente, a tratar con el cliente ayudando a valorar sus propuestas y a proponer otras nuevas, y desarrollando requisitos especificados por el mismo.

Dentro de este proyecto, he sido el encargado de desarrollar los principales componentes relacionados con la interfaz junto con las funcionalidades de los servicios que usaban estos elementos. Esto me ha permitido profundizar mucho en el desarrollo de interfaces Android, pero también, aprender a desarrollar la capa de negocio y conocer las formas de acceder a los servidores y a las bases de datos en Android. Además, aprovechando que he estado casi desde el principio del proyecto, he tenido la oportunidad de realizar pruebas de concepto y, decidir y hablar con el jefe de proyecto posibles soluciones a los problemas vistos en los diseños originales.

El proyecto se encuentra todavía en una fase temprana de desarrollo, aun así, creo que se está desarrollando correctamente de acuerdo a los patrones de diseño que mejor le vienen a un proyecto de estas características. Estoy convencido de que el software desarrollado se está elaborando usando técnicas que facilitarán cualquier mantenimiento o modificación futura.

6.2 Trabajo futuro

A corto plazo, aún queda mucho desarrollo hasta llegar al nivel de la aplicación de Windows Mobile actual. Aun así, según los desarrolladores van cogiendo experiencia el desarrollo se realiza de una manera más rápida y eficiente. También se notará una mejora en la velocidad al reutilizar componentes, gracias a la genericidad de los componentes desarrollados.

A largo plazo, la renovación del servidor y de la aplicación conseguirán un software estable y fácil de mantener con una vida útil larga. Además, el software final será un software abierto a mejoras con facilidad para la integración de nuevas tecnologías.

Referencias

[1] Xavier Albaladejo “Qué es SCRUM” https://proyectosagiles.org/que-es-scrum/

[2] Carlos Ble “Diseño ágil con TDD” http://librosweb.es/libro/tdd/

[3] http://www.elsi.es/ Motorola TC55 http://www.elsi.es/pda-industrial-terminal-movil/motorola-tc55-ref394.html

[4] Uncle Bob “The Clean Architecture” http://www.8thlight.com/ Agosto 2012 https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

[5] Jordan Schaenzle “The mock object approach to test-driven development” http://www.embedded.com/ Octubre 2012 http://www.embedded.com/design/prototyping-and-development/4398723/The-mock-object-approach-to-test-driven-development

[6] Miljenjo Cvjetko “What is Xamarin?” http://www.quora.com Marzo 2016 https://www.quora.com/What-is-XAMARIN-Is-it-a-framework-Or-compiler-Or-is-it-a-just-a-code-translator

[7] “TDD for Testers” http://www.ministryoftesting.com/ Agosto 2014 http://www.ministryoftesting.com/2014/08/tdd-testers/

[8] Anónimo “SOLID (object-oriented design)” https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Glosario

API	Application Programming Interface
Actividad	En Android se refiere normalmente a una pantalla
Apk	Android Application Package
Aserción	Acción y efecto de afirmar o dar por cierto algo, en programación son predicados que sirven para comprobar sentencias.
ERP	Enterprise Resource Planning
FTP	File Transfer Protocol
Release	Versión final que se entrega. Es aquella versión que normalmente, no es para el uso programador, sino que va dirigida al cliente
Trazabilidad	Serie de procedimientos que permiten seguir el proceso de evolución de un producto en cada una de sus etapas.

Anexos

A Xamarin Framework

Este anexo está centrado en las diferencias, ventajas y desventajas sobre Android por lo que es necesario un conocimiento previo de Android para entender la mayoría de los conceptos que se manejan.

Xamarin es la compañía encargada del Proyecto Mono [6] (.Net fuera de Windows), partiendo de esta idea, desarrollaron Xamarin Framework (conocido como Xamarin) que permite utilizar .Net para desarrollar apps para Android e iOS. Además, lo hace utilizando el código nativo de la plataforma, los accesos API y las interfaces de usuario con lo que conseguirá que no se diferencie el rendimiento entre una aplicación desarrollada con el código nativo de la plataforma, a uno desarrollado con C# a través de Xamarin.
Aparte de las diferencias en los lenguajes (no voy a entrar en las diferencias entre Java y C#), la forma de desarrollar código Android con Xamarin algunas ventajas de desarrollar para Android desde Xamarin son:

- Se puede reutilizar código (toda la capa de negocio si está bien separada) para desarrollo en otras plataformas.
- Las propiedades de las actividades se declaran al principio de la clase que las implementa, y no, todas juntas en el manifest. De esta forma, el desarrollador tiene las propiedades de la actividad directamente junto al código, facilitando su modificación. Será en tiempo de compilación, cuando el compilador de Xamarin traduzca estas propiedades y las inserte en manifest que lleva el apk, pero el desarrollador no tendrá que preocuparse por eso.
- Se pueden utilizar reglas lambda y delegados, que en Android no estaban presentes hasta la última API que ya incluye soporte para Java 8.
- Se pueden usar dll y complementos de otros desarrolladores, Xamarin incluye un centro de software donde se pueden encontrar complementos y componentes que se pueden utilizar directamente.

Pero no todo son ventajas, también podemos encontrar inconvenientes:

- Visual Studio no tiene buen soporte para Android, por lo que se perderán muchas funcionalidades qué si están en Android Studio, este problema se notará sobre todo a la hora de desarrollar las interfaces de usuario donde se comportará de una manera muy lenta y ofreciendo muy poca ayuda al desarrollador.
- Xamarin Studio no dispone de un gran soporte por parte de la comunidad, ni es tan completo en funcionalidades cómo Android Studio o Visual Studio.
- La integración de código existente en Java, no siempre funciona y, por tanto, pueden surgir problemas derivados de ello. Esto será muy importante a la hora de utilizar librerías creadas por los fabricantes de hardware como, por ejemplo, la librería que controla el lector de códigos de barras.
- Los procesos de compilación y debug son más lentos que en la plataforma nativa.
- Al poner una capa por encima de la API de Google, creamos un paso intermedio antes de la utilización de los componentes y aumentamos la probabilidad de que ocurran errores.

Athelia lo eligió como plataforma porque no hay diferencia en el rendimiento final ni en los resultados y, de esta manera, podía aprovechar la experiencia de los desarrolladores y el código existente en C#.

B Manual del programador

La aplicación de Servitrax es una aplicación muy grande que cuenta con muchos procesos y desarrollos complicados, además, cuenta con una complejidad de lógica de negocio muy alta debido a que ofrece soporte a una gran diversidad de funciones. Por tanto, para facilitar la entrada a nuevos programadores al proyecto, aparte de la documentación oficial del proyecto (documentación de tareas, presentación del proyecto, documentos de requisitos, …) mantenemos documentos donde se explica cómo desarrollar para Servitrax, cómo instalarse el entorno necesario para poder desarrollar y cómo utilizar componentes básicos de la aplicación como los explicados anteriormente. Por tanto, cualquier desarrollador que llegue al proyecto, antes de comenzar a programar deberá leerse estos documentos, instalarse el entorno y documentarse sobre el proyecto (requisitos, objetivos, partes, …).